Performance improvement with the INMOS Dx314 ANSI C Toolset

INMOS Limited

SGS-THOMSON MICROELECTRONICS

INMOS is a member of the SGS–THOMSON Microelectronics Group

72 TDS 354 00

October 1992

INMOS Document Number: 72 TDS 354 00

Contents

Preface

Host versions

The documentation set which accompanies the ANSI C toolset is designed to cover all host versions of the toolset:

- IMS D7314 – IBM PC compatible running MS–DOS
- IMS D4314 – Sun 4 systems running SunOS.
- IMS D6314 – VAX systems running VMS.

About this document

'Performance Improvement with the DX314 ANSI C Toolset'

This document provides advice about how to maximize the performance of the toolset. It brings together information provided in other toolset documents particularly from the *Language and Libraries Reference Manual.*

The document describes the layout of code and data in memory for programs developed with the ANSI C Dx314 Toolset. It then goes on to describe methods of improving code in order to:

- minimize the running time of the program;
- reduce the size of the program; either code or data or both.

Note: details of how to manipulate the software virtual through-routing mechanism are given in the *User Guide.*

About the toolset documentation set

The documentation set comprises the following volumes:

- *72 TDS 345 01 ANSI C Toolset User Guide*

  Describes the use of the toolset in developing programs for running on the transputer. The manual is divided into two sections; '*Basics*' which describes each of the main stages of the development process and includes a '*Getting started*' tutorial. The '*Advanced Techniques*' section is aimed at more experienced users. The appendices contain a glossary of terms and a bibliography. Several of the chapters are generic to other INMOS toolsets.

- *72 TDS 346 01 ANSI C Toolset Reference Manual*

  Provides reference material for each tool in the toolset including command line options, syntax and error messages. Many of the tools in the toolset are generic to other INMOS toolset products i.e. the occam and FOR-

TRAN toolsets and the documentation reflects this. Examples are given in C. The appendices provide details of toolset conventions, transputer types, the assembler, server protocol, ITERM files and bootstrap loaders.

- *72 TDS 347 01 ANSI C Language and Libraries Reference Manual*

  Provides a language reference for the toolset and implementation data. A list of the library functions provided is followed by detailed information about each function. Details are also provided about how to modify the run-time startup system, although only the very experienced user should attempt this.

- *72 TDS 348 01 ANSI C Optimizing Compiler User Guide*

  Provides reference and user information specific to the ANSI C optimizing compiler. Examples of the type of optimizations available are provided in the appendices. This manual should be read in conjunction with the reference chapter for the standard ANSI C compiler, provided in the *Tools Reference Manual.*

- *72 TDS 354 00 Performance Improvement with the DX314 ANSI C Toolset* (this document)

- *72 TDS 355 00 ANSI C Toolset Handbook*

  A separately bound reference manual which lists the command line options for each tool and the library functions. It is provided for quick reference and summarizes information provided in more detail in the *Tools Reference Manual* and the *Language and Libraries Reference Manual.*

- *72 TDS 360 00 ANSI C Toolset Master Index*

  A separately bound master index which covers the *User Guide, Toolset Reference Manual, Language and Libraries Reference Manual, Optimizing Compiler User Guide* and the *Performance Improvement* document.

Other documents

Other documents provided with the toolset product include:

- Delivery manual giving installation data, this document is host specific.
- Release notes, common to all host versions of the toolset.

occam and FORTRAN toolsets

At the time of writing the occam and FORTRAN toolset products referred to in this document set are still under development and specific details relating to them are subject to change. Users should consult the documentation provided with the corresponding toolset product for specific information on that product.

Documentation conventions

The following typographical conventions are used in this manual:

Bold type	Used to emphasize new or special terminology.
`Teletype`	Used to distinguish command line examples, code fragments, and program listings from normal text.
Italic type	In command syntax definitions, used to stand for an argument of a particular type. Used within text for emphasis and for book titles.
Braces { }	Used to denote optional items in command syntax.
Brackets [ ]	Used in command syntax to denote optional items on the command line.
Ellipsis . . .	In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.
|	In command syntax, separates two mutually exclusive alternatives.

1 Introduction

This document describes the layout in memory of C programs which have been built with the INMOS Dx314 ANSI C toolsets and suggests ways in which the performance of C programs may be improved.

This document assumes that the reader is familiar with the ANSI C toolset. The accompanying user documentation should be read in conjunction with this document.

Except where explicitly stated, all comments in this document apply to both the non-optimizing and the optimizing ANSI C compilers supplied with the Dx314 ANSI C toolsets. The transputer targets which this document applies to are the:

IMS T212, T222, T225, T400, T414, T425, T426, T800, T801, T805.

1.1 Transputer architecture

This document will not attempt to describe the transputer architecture. However, the particular points to note about the transputer, when considering performance maximization, are as follows:

- On-chip RAM.

  Each transputer has a part of its address space implemented as on-chip RAM, which means that it can be accessed very quickly. There is a noticeable penalty in accessing external RAM. Much of this document describes methods to ensure that best use is made of this on-chip RAM. Current transputer variants have either 2K or 4K bytes of on-chip RAM.

- Instruction prefixing.

  Transputers use a variable length instruction encoding, which is built up out of lots of single byte instructions. It is useful to minimize the size of these instructions, both to minimize the code space required, and to minimize the time taken to fetch the instructions from memory.

  In practice, the only instructions whose length can be easily controlled are those which access local variables; hence it is the layout of local variables which is important.

1.2 General issues

1.2.1 Space versus time

Most optimizations which are described are intended to minimize the running time of a program; this is known as optimizing for time. In certain circumstances it is required to minimize the size of a program; either code size, or data size, or both. This is known as optimizing for space. Often a particular optimization will produce an improvement in both space and time.

1.2.2 Processor classes and types

The ANSI C compiler can create code which can execute on many different types of transputer; these are known as transputer classes. This facility can be useful to build libraries which can be used for any transputer type. However, compiling for a particular transputer type will make best use of that transputer's particular instruction set, and therefore will make a program smaller and execute faster.

It is worth noting that you can create a library which contains, for example, both TA and T425 code. The linker will automatically select the most specific modules which exist in that library, depending upon the command line options supplied.

The rule to use is: always compile and link for the specific transputer type to get the best performance.

1.2.3 Full versus reduced libraries

Two versions of the C run-time library are provided: a full library and a reduced library. The full library provides all library functions documented in the *ANSI C Language and Libraries Manual*; the reduced library contains only those functions which do not require support from the host server, `iserver`.

For each library function, the function description in the *ANSI C Language and Libraries Manual* mentions if that function is not included in the reduced library: if you write code which calls any of these functions, and then attempt to link with the reduced libraries, you will get a linker error.

For programs which do not require host server support - for example, code installed in embedded systems, or code on processors which only communicate with neighboring processors, then linking with the reduced libraries is preferable as the size of the resulting linked unit will be smaller.

1.2.4 Start–up code

The runtime start-up code for configured systems is supplied in source form, so that you can tailor it to your exact requirements. For example, if your source code doesn't require heap or stack checking, then the relevant section of the start-up code may be removed.

Tailoring the start-up code can reduce the library overhead so that, for instance, the program may fit entirely in internal RAM.

Full details of the start-up code and how to modify it are given in the *ANSI C Language and Libraries Manual.*

1.3 Obtaining information

Various tools provide information which can be useful when improving performance.

Compiler information The compiler's `I` command-line option displays the number of bytes of code and static data in the module. The compiler's `P` command-line option produces a map file, which contains a map of the code and static data for the source file, together with a map of the local workspace for every function defined in the file.

Collector information The collector's `P` command-line option can be used to produce a text file which indicates the memory layout of each processor in the network. It also indicates the processor connectivity.

Memory map information The mapping tool, `imap`, will combine map files produced by the collector, linker and compiler into one map which gives a detailed description of the memory layout of each processor in the network.

`ilist` information The `ilist` program can examine any data file which is created by the INMOS toolsets, and display a decoded form of its contents. This may be useful if extra information is required which is not available by any of the previous methods.

2 Memory layout

For the purposes of this document, all diagrams of the transputer's address space have the most negative addresses at the bottom, and the most positive addresses at the top. (Remember that the transputer has a *signed* address space.) The *bottom* of the address space is the most negatively addressed memory location; and one memory address is *lower* than another if it has a more negative address.

As a transputer's memory map consists of areas of memory of widely varying speeds, correct use of the fastest memory is crucial in order to make programs run quickly. Usually, *this is the single most important factor in speeding up programs.*

The fastest area of memory is the on-chip RAM: this is located at the very bottom of the address space. As a general rule of thumb, a program's *stack* should be placed onto the on-chip RAM if possible. This is because the transputer's instruction encoding makes data accesses more frequent than instruction accesses, therefore greatest benefit is gained, if the data resides in fast memory. If there is space, then it is useful to put inner loops and other frequently used code subroutines into on-chip RAM too.

Some TRAMs, with memory of differing speeds, are constructed in such a way that the faster memory is always at a lower address. Thus the bottom 4K might be on-chip RAM, the next 32K might be 3-cycle external SRAM, followed by 2M of 4-cycle external DRAM. So the general rule is to put the more crucial parts of a program (code or data) at lower addresses.

2.1 Memory layout for configured programs

The usual way to build a program is to compile all the source files into object files, link all the object files into linked units, configure the linked units into a configuration data file, and then collect the configuration data file into a transputer executable file. A program built in this way is a *configured* program.

Figure 2.1 illustrates the default memory layout for a configured C program.

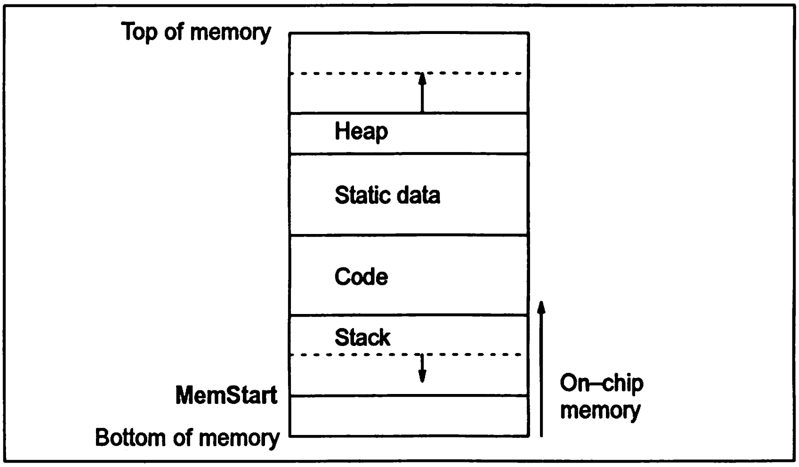

Figure 2.1 Default memory layout for a configured C program

This layout places the stack at the lowest address, followed by the code. This is normally the best ordering to obtain fastest program execution.

The sizes of the stack and heap sections must be specified explicitly in the configuration file. Take care not to specify too large a stack size: if this happens then it may be that the lowest-addressed part of the stack area, that which resides in on-chip memory, will never be used!

For example, if you specify a stack size of 6 Kbytes, but the program only actually uses 2 Kbytes, then the lower 4 Kbytes of the stack space will never be used (as the stack is a *falling* stack). If the stack area has been placed in the lowest memory, this means that the lower 4 Kbytes of memory – the on-chip RAM, will never be used.

When a program's stack requirement is substantially larger than the on-chip RAM, it may well be that, most of the time, the stack being used is off-chip. In this case, it might be more useful to move the code on-chip, particularly time-critical code sections, and leave the stack off-chip.

The configurer has several attributes which give greater control of the position and ordering of the different sections. Full details are given in the chapter entitled "*Advanced use of the configurer*" in the "*INMOS ANSI C Toolset User Guide*", supplied with the Dx314 toolset.

In summary, the attributes are:

`reserved` This is a processor attribute, which specifies an area of memory on a processor, which the configurer will not use when placing user and system processes.

`order` This is a process attribute which allows you to control the order in which the code, stack and heap sections for a given process are placed by the configurer.

`location` This is a process attribute which allows you to specify absolute addresses for the code, stack and heap sections for a given process, rather than have the configurer place them automatically.

2.2 Memory layout for non–configured programs

For single-processor programs, it is possible to skip the configuration stage and run `icollect` directly on the output of the linker. In this case, the collector's `T` command-line option must be used. A program built in this way is a *non-configured* program.

Figure 2.2 illustrates the default layout for a non-configured C program.

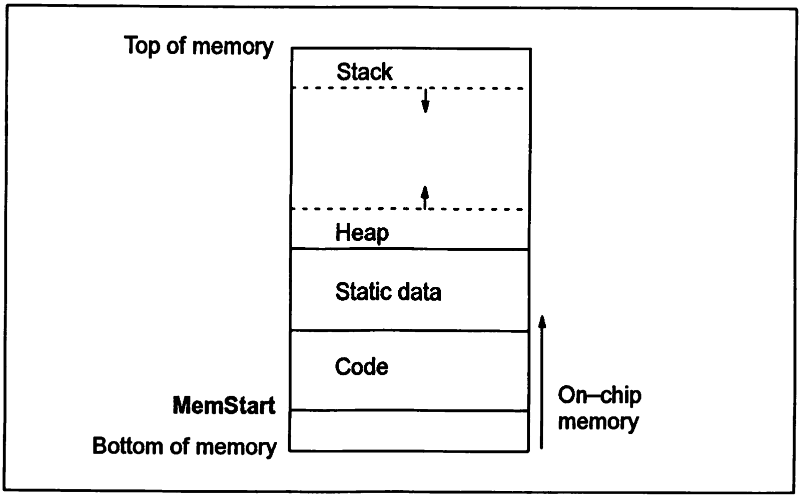

Figure 2.2 Default memory layout for a non–configured C program

The advantage of this layout is that it places no restriction (apart from the amount of available memory) upon stack size or heap size: the stack and heap can keep growing until there is no free memory left. The disadvantage of this layout is that it puts the stack into the slowest memory. As the stack is generally the most frequently accessed memory, this can have a serious impact on program speed.

For a non-configured program, it is possible to use `icollect`'s command line option S to place the stack in the lowest memory. Figure 2.3 illustrates the layout in this case.

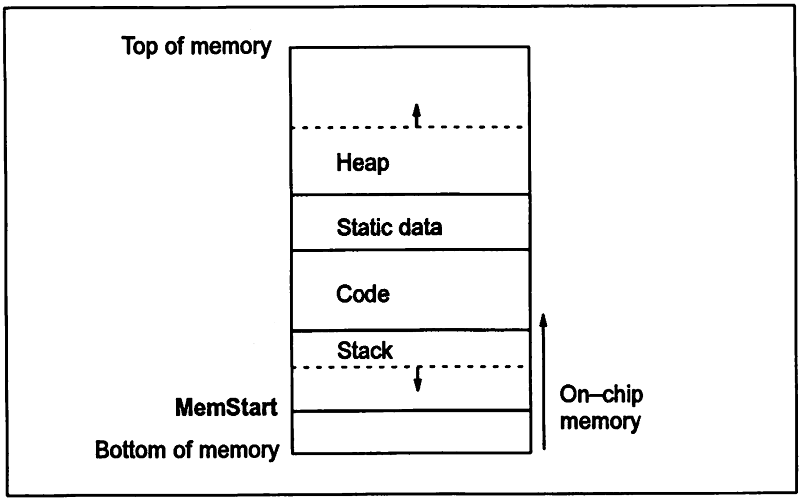

Figure 2.3 Memory layout for non–configured C program with `icollect s` option

The advantage of this layout is that it matches the guidelines given above for obtaining the fastest program execution. As the stack works downwards, the stack for the most nested routines will be in lower, and therefore faster, memory, which is what we desire. There is no restriction (apart from the amount of available memory) upon heap size.

2.3 Code layout

Within a file, code is generated for functions in the order in which they appear: the code for the first function in the file will be placed at the lowest address.

The linker allows a programmer to control the relative ordering of different modules in the linked object file. The output file will still be a single consecutive chunk of code, but the relative order of object files can be controlled. Primarily this is done by rearranging the order in which the files are listed on the command line. The linker inserts all separately compiled units into the output code file in the same order as they are encountered on the command line. It then adds library modules as necessary.

The linker provides finer control than this if required. This is done by means of `#section` directives in the linker's input file. By default, the compiler places all code in any compilation module into a 'code section' named "`text%base`". This may be overridden by use of the compiler's `#pragma IMS_linkage()`. If this pragma is used, the code section is named "`pri%text%base`". If the pragma is followed by a string parameter, that name is used for the code section.

For example:

```
#pragma IMS_linkage("fastcode")
```

names the current file's code section "`fastcode`".

The linker links all code modules in any particular named section, and then concatenates the sections. However, by naming different sections, a programmer can control the overall order. Normally, the linker places the section named "`pri%text%base`" at the beginning of the code, followed by "`text%base`", followed by any other code sections in an arbitrary order.

If the programmer supplies *any* `#section` directives in the linker's input file, the ordering is different. The linker places the first named section first, followed by the next named section, etc. Any sections which were not explicitly named are placed at the end. (**Note:** that the `#section` directive should be followed by the section name *without* enclosing quotes).

The map created by the mapping tool, `imap`, can be examined to confirm the relative placement of sections.

Note: that floating point support libraries used on `T4` series transputers are automatically placed into section "`pri%text%base`", so that they are more likely to be placed onto on-chip RAM.

2.4 Static data layout

The static data area comprises a local static area for each object file (or more specifically, each object file which uses static data) together with a module table. Figure 2.4 illustrates this.

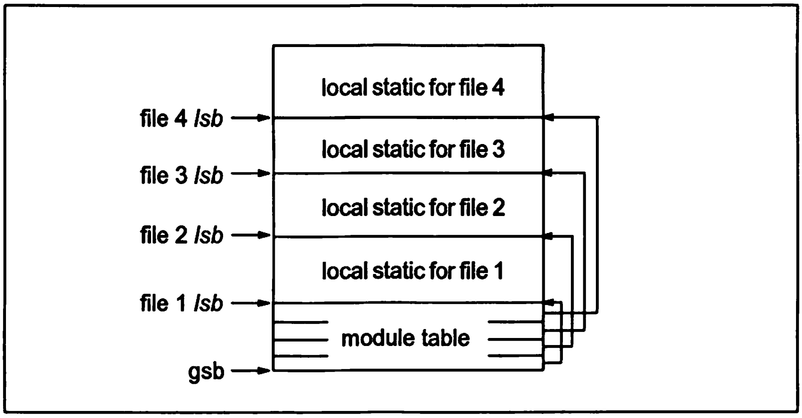

Figure 2.4 Static data layout

The module table contains an entry for every file with a local static area, which consists of a word containing a pointer to that file's local static area.

The base of the module table is called the global static base, or *gsb*.

2.4.1 Local static data layout

Usually, static data objects defined in a file are allocated space in that file's local static area. However, under certain conditions, a static data object may be placed in the text section (i.e. the section which contains the code) for that file (see section 2.4.2).

Local static data is allocated to increasing addresses in the local static area, in the same order as it appears in the source code.

The global static base (*gsb*), is passed as a hidden first parameter to every routine.

To access a piece of static data, the compiler loads the *gsb*, then does an indirect load to pick up the entry in the module table for the current file, this gives a pointer to the local static area (the local static base, or *lsb*). If the static data required is in the local static area, it may be accessed using the *lsb*; but if it is in another file's static area, then another level of indirection is required.

If a function makes frequent access to the local static area, then the *lsb* is cached into a temporary in local workspace before it is first used (usually, this is on entry to the function).

2.4.2 Constant static objects

If a static data object can be guaranteed to be non-modifiable, then the C compiler is sometimes able to allocate it in in the text section (i.e. the section which contains the code) for the file in which it is defined. The object must be non-modifiable, as the text section must be ROMable.

This can be useful as it can reduce the amount of memory required for that object: if the object is placed in the static data area then it must be initialized at program start-up and the value of the initializer is held in the text section. By allocating the object directly in the text section, no initializer is necessary. **Note:** that this will not reduce the size of the text section (and hence the size of the bootable file), but it will reduce the size of the static data area.

The exact conditions which must be satisfied for the object to be placed in the text section are:

1 The static data object must be declared as `const`.

2 The static data object must not be declared as `volatile`.

3 The static data object must have an initializer.

4 The initializer must contain no pointers to data or functions (absolute pointer values cannot be put into the text section as they are only known at run-time).

5 The static data object must not be externally visible (references to external objects have to know whether the object they are referencing is in the text section or the data section).

This can be useful if a program contains a very large table of constants or constant data.

For example:

```
static const char data[] = { 1, 27, 34, 52, ...
                             ...
                             ..., 5, 4, 0 };
```

will be allocated in the text section.

Note: that the conditions above require that the constant static data object must not be visible in any other files. This can be worked around by defining a pointer to the constant static object and making the pointer externally visible. For the above, we can define:

```
extern const char *datap = &data[0];
```

and then other files may access `data` indirectly through `datap`.

If you wish to ensure that a data object is *not* allocated in the text section then do not declare it as `const`.

2.5 Stack layout

Local workspace is allocated as a falling stack. It is not possible for the compiler to determine the maximum stack size; however, the `KS` option directs the compiler to insert a call to a stack checking routine on entry to every function. The stack checking routine will ensure that there is enough stack available to execute the function. (The optimizing C compiler will suppress the call to the stack checking routine for leaf[2] functions, when it can determine that enough stack is available.)

If stack checking is turned on, then at the end of a program the library routine `max_stack_usage` may be called; this returns the number of bytes of stack space used by the program. This value is the maximum value of stack size at each call to the stack checking function; so it must be treated with care if there are parts of the program running without stack checking.

As the runtime library never runs with stack-checking turned on, `max_stack_usage` always adds 150 words to the stack usage, as this allows for the largest stack required by the runtime library.

2. A 'leaf' function is a function that calls no other function.

2.6 Layout of structures

Layout of structures is very much dependent upon the particular compiler and the architecture of the target processor. The rules described here are very specific to the INMOS ANSI C compiler and the transputer, they should not be treated as general rules for laying out structures.

The compiler uses the following rules when laying out the fields within a structure:

1 C requires that structure fields are laid out in memory in the same order as they are specified in the source code.

2 `chars` (which are represented in one byte) may have any alignment.

3 `shorts` (which are represented in two bytes) are aligned on an even boundary.

4 word-sized or larger objects are aligned on a word boundary.

5 structures, unions and arrays are aligned on a word boundary.

`char` and `short` fields will be packed into the same word where possible, without breaking any of the above rules.

For example, on a 32-bit processor,

`struct d {`	
`char hid[8];`	The first byte of `hid` is on a word boundary (as the first byte of structure is on a word boundary), it occupies 8 bytes (2 whole words).
`unsigned short inuse;`	This occupies the lower two bytes of the following word.
`char flags1;`	This is packed into byte 2 of the same word as `inuse`.
`char flags2;`	This is packed into the upper byte of the same word as `inuse` and `flags1`.
`unsigned long tkey;`	This occupies the following word.
`unsigned short tfil;`	This occupies the lower two bytes of the following word.
`long npos;`	This has to be allocated on the next word boundary, so two bytes are left unused.
`unsigned short kmod;`	This occupies the lower two bytes of the following word.
`unsigned short kbhz;`	This is packed into the upper two bytes of the same word as `kmod` — 16–bit objects are placed at even addresses (rule 2), not word–addresses.
`unsigned short rmod;`	This occupies the lower two bytes of the following word.
`} structure;`	Two bytes are left unused.

Or, more graphically:

```
            Byte  0           1           2           3

     Word
     0            hid[0]      hid[1]      hid[2]      hid[3]
     1            hid[4]      hid[5]      hid[6]      hid[7]
     2            ◄──── inuse ────►       flags1      flags2
     3            ◄───────────── tkey ──────────────────►
     4            ◄──── tfil  ────►       ◄──── unused ────►
     5            ◄───────────── npos ──────────────────►
     6            ◄──── kmod  ────►       ◄──── kbhz   ────►
     7            ◄──── rmod  ────►       ◄──── unused ────►
```

If the structure fields are reordered, then a more efficient packing could be obtained.

```
     struct d {
       char hid[8];
       unsigned short inuse;
       char flags1;
       char flags2;
       unsigned long tkey;
       long npos;
       unsigned short kmod;
       unsigned short kbhz;
       unsigned short rmod;
       unsigned short tfil;
     } structure;
```

would give:

```
            Byte  0           1           2           3

     Word
     0            hid[0]      hid[1]      hid[2]      hid[3]
     1            hid[4]      hid[5]      hid[6]      hid[7]
     2            ◄──── inuse ────►       flags1      flags2
     3            ◄───────────── tkey ──────────────────►
     4            ◄───────────── npos ──────────────────►
     5            ◄──── kmod  ────►       ◄──── kbhz   ────►
     6            ◄──── rmod  ────►       ◄──── tfil   ────►
```

Note: the INMOS C compiler will generate more efficient code to load a `short` if it is word-aligned, so this new packing means that more code will be needed to access `tfil`, as it is no longer word-aligned. (Again, this is very dependant upon the way the INMOS ANSI C compiler currently handles structures.)

A general rule for obtaining the smallest structure size possible is to order the fields in increasing order of size.

2.7 Memory mapped devices

Memory-mapped devices can be accessed by declaring a structure which describes the registers of the device, and then declaring a pointer to the structure, which is initialized to the absolute address of the device.

For example, a memory mapped UART at absolute address `#40000000`:

```
typedef struct IODevice
  { int datareg;
    union
      { int controlreg;
        int statusreg;
      } u;
  } IODevice;
```

Then declare:

```
volatile IODevice *keyboard = (IODevice *)0x40000000;
```

Alternatively, you can use:

```
#define keyboard ((IODevice *)0x40000000)
```

For fastest access to the memory-mapped device, use the first method of declaring `keyboard`, and ensure that `keyboard` is a local variable. Access to static data is slower than access to local data, so for fast access, don't declare `keyboard` as a static variable; if you really want `keyboard` to be visible across multiple functions, then use the second (`#define`) method of declaring `keyboard`.

Then for example, to read the data register and write to the control register:

```
v = keyboard->datareg;
keyboard->u.controlreg = 1;
```

For a memory-mapped device at address zero, special care must be taken, as a cast of constant 0 to a pointer type is specially defined in C to produce the NULL pointer constant, which may or may not be zero. (On a 32–bit transputer it is 0 but on a 16-bit transputer, the NULL pointer constant has the value `0x8000`.)

To obtain a pointer to memory location zero, use:

```
union { int v; void *p; } ZeroPointer = { 0 };
IODevice *keyboard = ZeroPointer.p;
```

Note: that this method assumes that the size of `int` and pointer are equivalent (which is true on all current transputer C implementations); also, this declaration of `keyboard` cannot be initialized statically.

3 Improving code

This section describes some techniques which can be used to improve C code in terms of code speed, size, or both.

3.1 General optimizations

There are many standard program optimizations, such as common subexpression elimination, and loop invariant code motion, which are all applicable to transputer C programs.

The INMOS optimizing ANSI C compiler contains a global optimizer which performs a number of transformations, when optimization is selected using a command-line option. Further details of these are given in the *ANSI C Optimizing Compiler User Guide*.

3.2 Loop unrolling

Unrolling loops can speed up execution considerably. Take the following piece of C, a simple vector addition:

```
for (i = 0; i < 2000; i++)
  a[i] = b[i] + c[i];
```

The transputer performs addition in one cycle, whereas the loop overhead is approximately 18 cycles. To increase performance we can increase the number of adds per loop:

```
for (i = 0; i < 2000; i += 16)
  {
    int *const aslice = &a[i];
    int *const bslice = &b[i];
    int *const cslice = &c[i];

    aslice[0]  = bslice[0]  + cslice[0];
    aslice[1]  = bslice[1]  + cslice[1];
    aslice[2]  = bslice[2]  + cslice[2];
             .................
    aslice[14] = bslice[14] + cslice[14];
    aslice[15] = bslice[15] + cslice[15];
  }

/* do the last few ... */
for (i = i - 16; i < 2000; i++)
  a[i] = b[i] + c[i];
```

Obviously, loops can be opened out in any language, on any processor, and performance will tend to be improved at the expense of increased code size. However, opening loops out in slices of 16 has an additional benefit on the transputer, as optimal code with no prefix instructions is generated for each addition statement. Compare the code generated for the two statements:

```
                                             ldl  i
                                             ldl  b
                                             wsub
                                             ldnl 0
                                             ldl  i
  a[i] = b[i] + c[i];                        ldl  c
                                             wsub
                                             ldnl 0
                                             add
                                             ldl  a
                                             ldl  i
                                             wsub
                                             stnl 0


                                             ldl  bslice
                                             ldnl 15
                                             ldl  cslice
  aslice[15] = bslice[15] + cslice[15];      ldnl 15
                                             add
                                             ldl  aslice
                                             stnl 15
```

The second piece of code is just over half the size of the first, and the amount of loop overhead is reduced by a factor of 16.

3.3 Pointer update versus array subscripting

Some C programming styles tend to favour pointer update over array subscripting.

For example, take the following piece of C, a simple vector addition:

```
int *a, *b, *c;
for (i = 0; i < 2000; i++)
  a[i] = b[i] + c[i];
```

A C programmer may more naturally write this as

```
for (i = 0; i < 2000; i++)
  *a++ = *b++ + *c++;
```

This removes the need for three array subscriptions at the expense of updating three pointers. On some architectures, the pointer increment comes for free, and so the second version would run more quickly. However, on the transputer, it is considerably more expensive to update the pointer than to perform the array subscription, and the second version actually takes 30 % more time than the first.

3.4 optimizing switch statements

The ANSI C compiler will attempt to make a `switch` statement as fast as possible, assuming that all the values are equally likely, and may use a combination of techniques to select the correct branch. This compiler uses a combination of jump tables, binary switches and explicit tests, depending upon the relative values and density of the target values (i.e. whether there are any 'gaps').

A combination of `if` statements and a `switch` may be the best solution where a few values are particularly common, but where there may be many other possibilities.

```
temp = complicated_expression;

if (temp == most_frequent_value)
  ...  process most_frequent_value
else if (temp == next_most_frequent value)
  ...  process next_most_frequent_value
else
  switch (temp)
    {
      case infrequent_value_1:
        ...  process infrequent_value_1
        break;
      case infrequent_value_2:
        ...  process infrequent_value_2
      ...  etc.
    }
```

3.5 Use ANSI function prototypes

As well as providing a certain amount of type-checking, the use of ANSI function prototypes can improve the code generated by the compiler.

In the absence of a function prototype, actual parameters have the default argument promotions performed on them: `char` and `short` arguments are widened to `int`; and `float` arguments are widened to `double`.

On entry to a function declared in the old (Kernighan and Ritchie) style, `char` and `short` formal parameters are narrowed down from `int` to the appropriate type, and `float` formal parameters are narrowed from `double`.

For example:

```
func(f1)
float f1;
{ ...
}

g()
{  float f2;
   ...
   func(f2);
}
```

Here `f2` is widened from `float` to `double` at the call to `func`; and immediately narrowed back down from `double` to `float` on entry to `func`.

If the function is defined in prototype-format, actual parameters are converted directly to the type of the formal parameters, as if by assignment.

So, by changing the function definitions to prototype-format in the above example,

```
void func(float f1)
{ ...
}

void g(void)
{ float f2;
  ...
  func(f2);
}
```

Now `f2` is passed as an argument without any type conversion at all.

3.6 Use floating point in single precision where possible

On the transputer, single-precision floating-point arithmetic is faster than double-precision, so where the greater accuracy of double-precision is not required, it can be beneficial to use single-precision.

In Kernighan and Ritchie C, floating-point arithmetic is always performed in double precision, for example to add two `float` variables, each is widened to `double`, the addition is performed, and then the result is narrowed back down to single-precision.

In ANSI C, however, floating-point arithmetic upon single precision values is performed in single precision: two `floats` are added in single precision. Therefore, only declare `double` variables when you really want double precision arithmetic.

The Dx314 toolset also provides a set of single-precision mathematical functions. The prototypes for these functions are declared in the header file `mathf.h`, the names of the functions are formed by appending an `f` onto the name of the corresponding double-precision function. For instance, `cos` performs cosine in double precision, so `cosf` performs cosine in single precision.

3.7 Short integers

On the current range of 32-bit transputers (see the start of chapter 1 for a list), loading and storing `short` integers is much less efficient than loading or storing `int`, so use `int` wherever possible.

A genuine use for `short` integers is for saving space in structures or arrays, as they are only two bytes long, whereas `int`s are four bytes long (on a 32-bit transputer).

Care must be taken when `short`s are packed into structures as the compiler can generate better code to load a word-aligned `short` field of a structure than a non-word-aligned one. This is a tradeoff: the space saved by packing the structure versus the extra code introduced to extract the packed fields; you must decide which is more critical.

3.8 Chars

By default, `char`s are unsigned: the ANSI standard leaves this as implementation defined, and provides a new type `signed char` for explicitly signed characters.

However, many UNIX C programs expect `char` to be signed by default, so the compiler provides a command-line option, `FC`, which directs the compiler to treat the type `char` as signed.

Care must be taken in the use of `FC` and `signed char`, as loading and storing a signed `char` is less efficient than loading and storing an unsigned `char`.

3.9 Use ANSI C `mem..` functions

ANSI C provides a number of memory manipulation functions as standard library functions. These routines have been carefully coded in the libraries to execute quickly on the transputer.

- `memcpy` copies a block of memory. The compiler will inline a call to `memcpy` as a block move instruction.
- `memmove` also copies a block of memory, but correctly handles an overlapping source and destination. It uses block move to copy the non-overlapping parts.
- `memset` is used to initialize a block of memory to a particular value. It uses the block move instruction to copy ever-increasing blocks of memory.

- `memcmp` is used to compare two blocks of memory; it uses a word-at-a-time comparison.
- `memchr` is used to search a block of memory for a given byte value.

3.10 Keep variables as local as possible

This section mainly relates to the non-optimizing C compiler: the INMOS optimizing ANSI C compiler uses a sophisticated "variable liveness" algorithm to deduce where to allocate variables in local workspace. However, keeping variables as local as possible can, in some cases, also help the optimizing C compiler, and it is, of course, good programming practice.

The non-optimizing C compiler allocates local workspace to variables by their scope.

More nested variables are allocated at smaller workspace offsets.

If two variables are in scope at the same time, they are allocated disjoint workspace slots. The compiler estimates a run-time usage for each local variable and allocates the variable with the greater estimated usage at a lower workspace position.

The run-time usage is currently calculated as follows:

- Every load or store of the variable counts as one use
- Every load or store within a loop counts as eight uses
- Every load or store within the inner of two nested loops counts as 64 uses, etc.
- If the variable is declared with a `register` storage class specifier, fifty uses are added to the sum of the load and store uses.

To reduce the size of workspace, it is best to keep variable declarations as local as possible to the code which uses them. For example, the compiler allocates two words of workspace for:

```
{ int a, b;
  ... use a
  ... use b
}
```

but only one word of workspace for

```
{ int a;
  ... use a
}
{ int b;
  ... use b
}
```

3.11 Access to static data

Access to local static data is less efficient than access to local automatic data.

Access to another file's static data is less efficient than access to the current file's static data.

Table 3.1 lists the code sequences used to load an integer variable *n* from the different areas of data.

Data area	Code sequence to load integer variable *n* from data area
local (automatic)	*ldl n*
static defined in this file	*ldl lsb; ldnl n*
static defined in another file	*ldl lsb; ldnl n.ptr; ldnl 0*

Table 3.1 Accessing static data

It is therefore recommended that you:

- make data automatic where possible,
- define static data in the file which accesses it most.

3.12 Large objects

Automatic arrays and structures are always allocated in local workspace. This means that the workspace can get quite large, which is undesirable if you are trying to fit it onto on-chip RAM.

To avoid placing a large object (array or structure) in workspace,

1. Put the large object in static space and declare a local pointer to it. This gives good efficiency of access, but unfortunately it is not re-entrant: only one copy of the object exists no matter how many times the function using it is called; also, it means that the object exists throughout the entire lifetime of the program, rather than just when the function which uses it is called.
2. Use `malloc` to obtain space on the heap for the object. Again, this gives good efficiency of access, and preserves re-entrancy. The memory used by the object can be freed up when no longer needed using `free`. The disadvantage is that calls to `malloc` and `free` can be time consuming.

A file's local static area is allocated in the order in which the static data definitions appear in the source code; to reduce the number of prefixes required to access an item of static data, it is best to order the static data definitions so that smaller objects appear first.

3.13 Built–in, inlined routines

The Dx314 ANSI C compiler supports a number of built-in routines. Any calls to these routines will be compiled directly inline where possible.

Each of these routines have prototypes in an INMOS-supplied header file. The automatic inlining will only happen if this header file has been `#included` in the user source.

Each of the built-in routines is designed to allow access to a transputer instruction which is not directly accessible from the C source level. Table 3.2 lists the routines and the instructions they support.

Function	Instruction supported	Header file
`BlockMove`	*move*	`misc.h`
`BitCnt`	*bitcnt*	`misc.h`
`BitCntSum`	*bitcnt*	`misc.h`
`BitRevNBits`	*bitrevnbits*	`misc.h`
`BitRevWord`	*bitrevword*	`misc.h`
`Move2D`	*move2dall*	`misc.h`
`Move2DNonZero`	*move2dnonzero*	`misc.h`
`Move2DZero`	*move2dzero*	`misc.h`
`CrcByte`	*crcbyte*	`misc.h`
`CrcWord`	*crcword*	`misc.h`
`DirectChanIn`	*in*	`channel.h`
`DirectChanInChar`	*in*	`channel.h`
`DirectChanInInt`	*in*	`channel.h`
`DirectChanOut`	*out*	`channel.h`
`DirectChanOutChar`	*outbyte*	`channel.h`
`DirectChanOutInt`	*outword*	`channel.h`
`ProcGetPriority`	*ldpri*	`process.h`
`ProcTime`	*ldtimer*	`process.h`
`ProcReschedule`	–	`process.h`

Table 3.2 Inlined functions

Further information on these routines may be found in the "*ANSI C Language and Libraries Manual*" supplied with the Dx314 toolset.

3.14 Channel communication

The normal channel input and output routines in the C run-time library (ChanIn, ChanInChar, ChanInInt, ChanOut, ChanOutChar and ChanOutInt) run on top of a virtual channel system to enable interactive debugging, and to allow the virtual configuration system to work. However, the virtual I/O is slower than direct channel I/O, and so if it is not required, and

- interactive debugging is not required, and
- the channel is either not declared in the configuration description file, or is placed on a specific hardware link in the configuration description file

then it is possible to replace calls to the normal channel input and output routines with calls to the 'direct' channel input and output routines: (DirectChanIn, DirectChanInChar, DirectChanInInt, DirectChanOut, DirectChanOutChar and DirectChanOutInt).

Each of the direct routines has an equivalent prototype to the corresponding normal routine, all that changes is the name of the routine.

To enable rapid switching between interactively-debuggable and fast code, something like the following could be placed in a header file:

```
#include <channel.h>
#ifdef FAST_IO
#define ChanIn(C,P,L)  DirectChanIn((C),(P),(L))
....
#endif
```

then all calls to ChanIn will be converted into calls to DirectChanIn when FAST_IO is defined.

3.15 External function definitions

Calling functions defined in another file is less efficient than calling functions defined in the same file as the external call has to go through a stub.

For example:

```
extern function();
...
function();
```

is compiled as:

```
               ...
               call function_stub
               ...
function_stub: j function
```

But if **`function`** is defined in the current file, the compiler will call it directly. It is possible to avoid all external calls by compiling the whole program in one go: for example, if a program consists of two files **`file1.c`** and **`file2.c`**, then create a file **`all.c`** which includes **`file1.c`** and **`file2.c`**, i.e.

```
#include "file1.c"
#include "file2.c"
```

and compile **`all.c`** instead.

(**Note**: that this trick doesn't work if there are name clashes between local static items defined in **`file1.c`** and **`file2.c`**.)

4 Other features

This section describes other features of the ANSI C compiler which may be useful when compiling C code.

4.1 Arithmetic right shift of signed integers

The ANSI C standard states that if a signed integer is shifted right *n* bits, then the resulting value of the upper *n* bits of the integer is undefined. For the transputer, the ANSI C compiler will generate a *logical* right shift in these situations. Many C programmers, however, assume an *arithmetic* right shift, and consequently their code breaks.

The Dx314 ANSI C compiler provides a command line option, `FS`, which directs the compiler to generate arithmetic right-shifts on signed integers.

Right-shifts on *unsigned* integers are always logical shifts.

4.2 Signedness of plain chars

As mentioned in section 3.8, plain `chars` are unsigned by default. As some existing C code assumes that plain `chars` are signed, the compiler provides a command-line option, `FC`, which directs the compiler to treat the type `char` as signed.

5 Running benchmarks

5.1 General rules

The following are very general rules for running benchmarks. They may not apply to a particular benchmark.

- If using the optimizing compiler, compile the program at the highest optimization level (O2).
- Compile the program *without* debugging information (i.e. without the G command line option). Enabling debug information can marginally increase the size of, and decrease the speed of code.
- Choose the most restricted startup code suitable to the needs of the program.
- Order the list of object files presented to the linker so that the most frequently executed files appear first on the list.
- For non-configured programs, use the collector's S option to place stack on-chip; and the M option to specify the memory size in advance. Make the size of the stack (the parameter to the collector's S option) as small as possible.
- For configured programs, use the configurer's default segment ordering (which will place the stack in internal RAM), and make the size of the stack segment as small as possible.

Index

Customer Checklist for IMS D0314

Ring binder 1 containing:	
Shrink–wrapped documentation	
ANSI C Toolset User Guide	
ANSI C Optimizing Compiler User Guide	
Performance improvement with the INMOS DX314 ANSI C Toolset	
User Group Information	
User Group Enrolment Form	
User Groups Addresses	
Software problem report form	
Address Sheet	

Ring Binder 2 containing:	
Shrink–wrapped documentation	
ANSI C Toolset Reference Manual	
Address Sheet	
ANSI C Toolset Handbook	

Ring Binder 3 containing:	
Shrink–wrapped documentation	
ANSI C Language and Libraries Reference Manual	
Address Sheet	
ANSI C Toolset Master Index	

Customer Checklist for IMS D4314

4–user Licence Terms and Conditions and Product Registration Form	
Ring binder containing:	
Shrink–wrapped documentation	
IMS D4314 delivery manual	
Release notes	
Address sheet	
Software problem report form	
1 x D4314 Sun data cartridge	

Customer Checklist for IMS D4300A

4–user Licence Terms and Conditions and Product Registration Form	
Ring binder containing:	
Shrink–wrapped documentation	
IMS D4300A Sun-4 Inquest Toolset delivery manual	
Inquest Toolset Debugger Tutorial	
Inquest Toolset User Manual	
Release notes	
Address sheet	
Software problem report form	
1 x D4300A QIC data cartridge	